20MCA 132 Object Oriented Programming Lab

CO3: Implement OOPs concepts to effectively solve near to real time problems.

Method Overloading

- Method Overloading is a feature of java that allows a class to have more than one method having the same name.
- Java can distinguish between overloaded methods with different method signatures.
- That is in an overloaded method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

- For eg:
  void add(int x, int y)
  void add(int x, int y, int z)

2. Data type of parameters.

- For eg:
  void add(int x, int y)
  void add(int a, float b)

3. Sequence of Data type of parameters.

For eg:

void add(int a, float b)
void add(float a, int b)

- In java, method overloading is not possible by changing the return type of the method only.
- For example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example.
- This will throw compilation error.

- That is
  - int add(int, int)
  - float add(int, int)
  - The above two statements will throw compilation error.

Example 1:- Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class Adder
{
        int add(int a,int b)
        {
                return a+b;
        }
         int add(int a,int b,int c)
        {
                return a+b+c;
        }
}
class TestOverloading1
{
        public static void main(String args[ ])
        {
          Adder obj1=new Adder( );
         int s1=obj1. add(10,20);
         int s2=obj1.add(10,20,30);
         System.out.println('Sum 1="+s1);
         System.out.println("Sum 2="+s2);
         }
}
```

Output

Sum1 =30
Sum 2=60

Example 2:- Overloading –Difference in data type of parameters

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder2
{
         int add(int a, int b)
        {
```

```
                return a+b;
        }
        double add(double a, double b)
        {
        return a+b;
        }
}
class TestOverloading2
{
        public static voi d main(String[] args)
        {
          Adder2 obj2=new Adder2( );
          int s1=obj2. add(10,20);
          int s2=obj2.add(12.3,12.6);
         System.out.println('Sum1="+s1);
         System.out.println("Sum2="+s2);
         }
}
```

Output

Sum1 =30
Sum 2=24.9

Example 3:- Overloading- change the sequence of data type of arguments

```
class student
{
         void Identity(String name,  int id)
        {
                System.out.println("Name 1:"+ name +" "+"Id :"+ id);
        }
         void Identity(int id, String n)
        {
                 System.out.println("Name2:"+ n +" "+"Id :"+ id);
         }
}
class overloading3
{
```

```
public static void main (String args[ ])
 {
   student stu = new student();
    stu.Identity("Mohit", 1);
    stu.Identity(2, “Ram");
   }
}
```

Constructor Overloading

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
class StudentData
{
        int id;
         String name;
         int age;
        // creating default constructor
        StudentData()
         {
                id = 100;
                name = "New Student";
                age = 18;
        }
         //creating two arg constructor
         StudentData(int i,String n)
        {
           id = i;
           name = n;
          }
```

```
    //creating three arg constructor
    StudentData(int i,String n,int a)
    {
        id = i;
        name = n;
        age=a;
    }
    void display()
    {
    System.out.println(id+" "+name+" "+age);
    }
}
class Student
{
    public static void main(String args[ ])
    {
    StudentData s1 = new StudentData( );
    StudentData s2 = new StudentData(111,"Karan");
    StudentData s3 = new StudentData(222,"Aryan",25);
        s1.display();
        s2.display();
        s3.display( );
     }
}
```

Inheritance

- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.
- The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).
- extends Keyword
    - **extends** is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

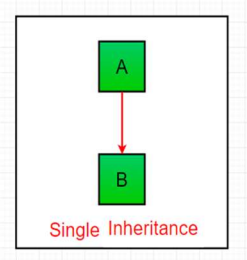
- **The Syntax is**

```
class base_class
{
 ..... .....
}
class sub_class extends base_class
{
 ..... .....
}
```

- The **extends keyword** indicates that we are making a new class that derives from an existing class.
- The meaning of "extends" is to increase the functionality.

Types of Inheritance in Java

1. **Single Inheritance :** In single inheritance, subclasses inherit the features of one super class. In image below, the class A serves as a base class for the derived class B.

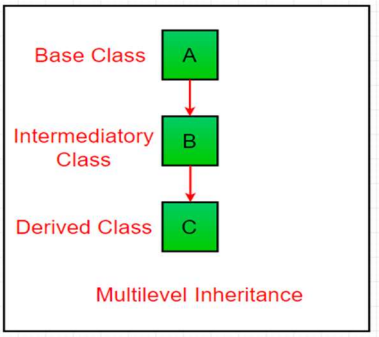


Example program for single level inheritance

```
class Rectdemo
{
int le,be;
void getval(int a,int b)
{
        le=a;
        be=b;
}
int rectArea( )
{
        return le*be;
}
```

```
}
class Triangle extends Rectdemo
{
        int b,h;
        float t;
void  getdata(int q,int r)
{
        b=r;
        h=q;
}
float triArea( )
{
        t=(float)le/2*b*h;
        return (t);
}
}
class single_inher
{
public static void main(String args[])
{
        Triangle Tr=new Triangle();
        Tr.getval(40,8);
        Tr.getdata(10,6);
        System.out.println("Area of Rectangle:"+  Tr.rectArea());
        System.out.println("Area of Triangle:"+Tr.triArea());
}
}
```

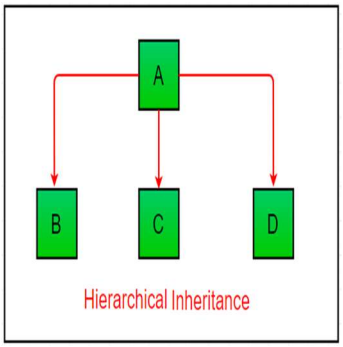
2. Multilevel Inheritance : In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



Syntax is:

```
class A
{
  .............
}
class B extends A
{
   ......................
}
class c extends B
{
  .....................................
}
```

3. Hierarchical Inheritance : In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.

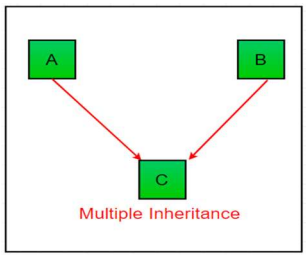


Syntax is:

```
class A
{
  .............
}
```

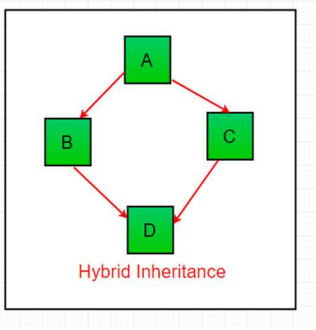
```
class B extends A
{
  .....
}
class c extends A
{
  .....
}
class D extends A
{
  .....
}
```

4. Multiple Inheritance : In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



class C extends A,B – this statement is invalid

4. Hybrid Inheritance : It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Example java program for multilevel inheritance- calculate the students total mark

```
class stud_details
{
    private int sno;
    private String sname;
    void setstud(int no,String name)
        {
                sno = no;
               sname = name;
         }
        void putstud()
        {
              System.out.println("Student No : " + sno);
              System.out.println("Student Name : " + sname);
        }
}
class marks extends stud_details
{
      protected int mark1,mark2;
      public void setmarks(int m1,int m2)
      {
            mark1 = m1;
            mark2 = m2;
      }
      public void putmarks()
      {
        System.out.println("Mark1 : " + mark1);
         System.out.println("Mark2 : " + mark2);
      }
}
```

```
class finaltot extends marks
{
        private int total;
         void calc()
        {
                total = mark1 + mark2;
        }
        public void puttotal()
        {
                 System.out.println("Total : " + total );
         }
}

class student
{
public static void main(String args[])
      {
            finaltot f = new finaltot();
            f.setstud(100,"Nithya");
            f.setmarks(78,89);
            f.calc();
            f.putstud();
            f.putmarks();
            f.puttotal();
      }
}
```

Super keyword in Java

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever we create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

The use of super keyword

1) To access the data members of parent class when both parent and child class have member with same name.

Example 1 for super keyword

```
class Superclass
 {
        int num = 100;
 }
 class Subclass extends Superclass
 {
        int num = 110;
        void printNumber()
        {
                System.out.println("First number="+super.num);
                System.out.println("second number="+num);
        }
}
class example_super
{
public static void main(String args[ ])
{
        Subclass obj= new Subclass( );
        obj.printNumber();
}
}
```

Output

```
First number=100
Second number=110
```

2) Invoke the parent class constructor.

- The keyword "super" is used to invoke the super class's constructor from within subclass's constructor.
- The statement super must be the first line of child class constructor. Calling a parent class constructor's name in the child class causes syntax error.

- **Example 2:- calling a default constructor using super**

```
class Person
{
        Person()
        {
        System.out.println("Person class Constructor");
```

```
        }
}
 /* subclass Student extending the Person class */
class Student extends Person
{
        Student()
        {
       // invoke or call parent class constructor
            super();
          System.out.println("Student class Constructor");
        }
}
class Test
{
  public static void main(String args[ ])
  {
     Student s = new Student();
  }
}
```

- **Example 3:- calling a parameterized constructor using super**

```
class Person
{
        int id;
        String name;
        Person(int i,String n)
        {
                id=i;
                name=n;
        }
}
class Emp extends Person
{
        float salary;
        Emp(int i,String n,float s)
        {
        super(i,n);
        salary=s;
```

```
        }
void display()
 {
System.out.println(id+" "+name+" "+salary);
}
}
class Test
{
public static void main(String args[ ])
{
        Emp e1=new Emp(1,"ankit",45000f);
        e1.display();
}
}
```

Method overriding

- Declaring a method in **sub class** which is already present in **parent class** is known as method overriding.
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- In overriding, method of both class **must** have **same name** and equal number of **parameters with similar datatype**.
- They have same method signature in sub classes with different method body.
- Constructors cannot be overridden.
- Example program1

```
// Base Class
class A
{
        int num1,num2;
        A( int a, int b)
        {
                num1=a;
                num2=b;
        }
        void show()
        {
```

```
        System.out,println("Number1="+num1);
        System.out,println("Number2="+num2);
        }
}
//Derived class
class B extends A
{
        int num3;
        B(int a, int b, int c)
        {
                super(a,b);
                num3=c;
        }
        void show( )
        {
                System.out,println("Number3="+num3);
        }
}
class test_override1
{
        public static void main(String args[ ])
        {
                B ob=new B(10,20,30);
                ob.show( );
        }
}
```

Output

Number3=30

- In method override the version of a method that is executed will be determined by the object that is used to invoke it.
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.
- In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

- We can call parent class method in overriding method using super keyword.

```
// Base Class
class A
{
        int num1,num2;
        A( int a, int b)
        {
                num1=a;
                num2=b;
        }
        void show()
        {
        System.out,println("Number1="+num1);
        System.out,println("Number2="+num2);
        }
}
//Derived class
class B extends A
{
        int num3;
        B(int a, int b, int c)
        {
                super(a,b);
                num3=c;
        }
        void show( )
        {
                super.show( );
                System.out,println("Number3="+num3);
        }
}
class test_override1
{
        public static void main(String args[ ])
        {
                B ob=new B(10,20,30);
                ob.show( );
```

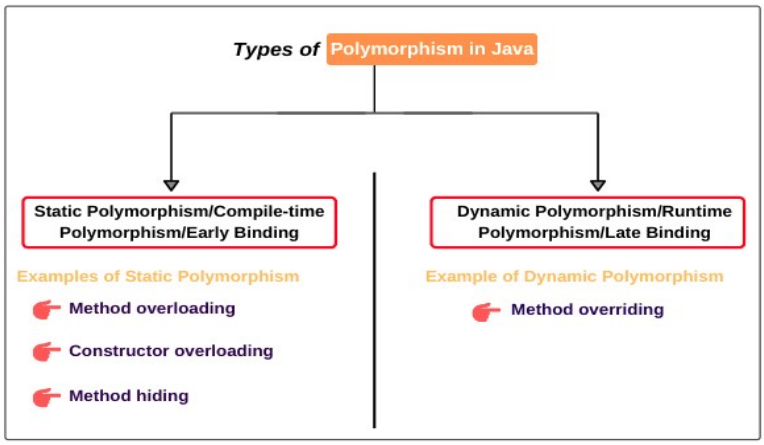
```
        }
}
```

Output

```
Number1=10
Number2=20
Number3=30
```

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- Polymorphism is derived from 2 Greek words: poly and morphs.
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.



- Static Polymorphism
  - static polymorphism is achieved through method overloading.
  - Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters.
  - At compile time, Java knows which method to invoke by checking the method signatures.
  - So, this is called compile time polymorphism or static binding.

- Method Overloading on the other hand is a compile time polymorphism- example program.

- **Example program**

```
class Overload
 {
        void demo (int a)
        {
        System.out.println ("a: " + a);
        }
        void demo (int a, int b)
         {
        System.out.println ("a and b: " + a + "," + b);
        }
        double demo(double a)
        {
        System.out.println("double a: " + a);
        return a*a;
         }
 }
class Poly1
        {
                public static void main (String args [])
                {
                Overload Obj = new Overload();
                Obj .demo(10);
                Obj .demo(10, 20);
                double result = Obj .demo(5.5);
                System.out.println("O/P : " + result);
                }
        }
```

Output

a: 10

a and b: 10,20

double a: 5.5

O/P : 30.25

- Here the method demo() is overloaded 3 times:
  - first method has 1 int parameter,
  - second method has 2 int parameters and
  - third one is having double parameter.
- Which method is to be called is determined by the arguments we pass while calling methods.
- This happens at compile time so this type of polymorphism is known as compile time polymorphism.

- **Dynamic Polymorphism in Java**
  - The type of polymorphism which is implemented dynamically when a program being executed is called as dynamic polymorphism.
  - The dynamic polymorphism is also called run-time polymorphism or late binding.
  - Method overriding is one of the ways in which Java supports Runtime Polymorphism.
  - Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
  - In method overriding the method must have the same name as in the parent class and it must have the same parameter as in the parent class.
  - If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

- Method overriding occurs in two classes that have inheritance.

- **Dynamic method dispatch**
  - Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime.
  - This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to.
  - In simple words the type of object which it referred determines which version of overridden method will be called.
  - It allows subclasses to have common methods and can redefine specific implementation for them.
  - This lets the superclass reference respond differently to same method call depending on which object it is pointing.

- For example,

```
Base b=new Base();
Derived d=new Derived();
b=new Derived();      // Base class reference refer to derived class
                                  // object  that is upcasting
```

  - Here Base is the parent class and b is the object of this class.
    - Derived is the child class and d is the object of this class.
  - In the third line of the program segment, the parent class reference variable 'b' refers to a child class object than it is called upcasting and using this technique dynamic method dispatch perform.
  - Example program

```
class Rectangle
{
   int l,b;
   int a;
   Rectangle( )
   {
         l=5;
         b=30;
    }
   void area()
   {
       a=l*b;
       System.out.println("Area of rectangle: "+a);
   }
}
class Square extends Rectangle
{
  void area()   //overridden method
  {
   a=l*l;
    System.out.println("Area of square: "+a);
     }
}
class Triangle extends Rectangle
{
    void area()     //overridden method
    {
        a=(b*l)/2;
       System.out.println("Area of triangle: "+a);
     }
}
 class Calculation
```

```
{
        public static void main(String args[])
        {
        Rectangle r=new Rectangle();
        r.area();
        r=new Square();  //superclass reference referring to subclass
                        // Square object so,at run time it will call Square
                                //area()
        r.area();
        r=new Triangle();    //superclass reference referring to subclass
                                //Triangle object so, at run time it will call triangle
                                //area()
        r.area();
      }
}
```

Output

Area of rectangle: 150

Area of square: 25

Area of triangle: 75

- At first, inside of the main method we declared the r as the object of Rectangle class.
- The object r will call the Rectangle class area() method.
- r=**new** Square(); this statement treated r as a reference variable of the class Rectangle and it is converted as an actual object of Square class.
- On the next line where we called the area( ) method on r.
- The java compiler verifies that indeed Rectangle class has a method named area( ), but the java runtime notices that the reference is actually an instance of class Square.
- So it calls Square class area( ) method.
- The same procedure is repeated with Triangle class also.

abstract keyword in java

- 'abstract' keyword is used to declare the method or a class as abstract.
- A class which contains the abstract keyword in its declaration is known as an abstract class.
  - If a class is declared abstract, it cannot be instantiated.
  - An abstract class is a restricted class that cannot be used to create objects.
  - A class which is not abstract is referred as Concrete class.
  - Example,

```
abstract class A
{
        //body of the abstract class
}
```

Here A is an abstract class

- Abstract classes may or may not contain *abstract methods*,
- But, if a class has at least one abstract method, then the class must be declared abstract.
- If we want a class to contain a particular method but the actual implementation of that method to be determined by child classes, then we declare the method in the parent class as an abstract.
- The abstract keyword is used to declare the method as abstract.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semicolon (;) at the end.
- For example,

```
abstract class Employee
{
        abstract void work( );
}
```

- The role of an abstract class is to contain abstract methods. However, it may also contain non-abstract methods. The non-abstract methods are known as concrete methods.

- For example,

```
abstract class MyClass
{
public void disp()
{
        System.out.println("Concrete method of parent class");
}
 abstract public void disp2();
}
```

- ✓ **Complete example for abstract**

```
// abstract class
abstract class Shape
{
  // abstract method
  abstract void sides();
}
class Triangle extends Shape
{
  void sides()
  {
    System.out.println("Triangle shape has three sides.");
  }
}
class Pentagon extends Shape
{
  void sides()
  {
    System.out.println("Pentagon shape has five sides.");
  }
}
class abstract_demo
{
   public static void main(String[] args)
   {
      Triangle obj1 = new Triangle();
```

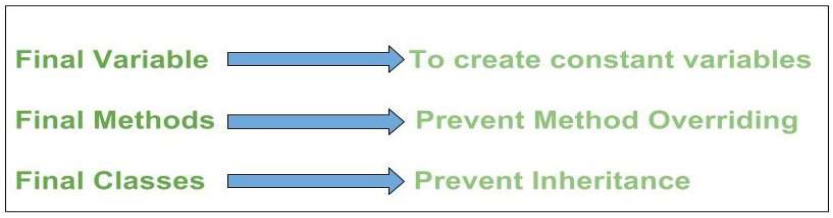
```
        obj1.sides();
        Pentagon obj2 = new Pentagon();
        obj2.sides();
    }
}
```

Output

Triangle shape has three sides.
Pentagon shape has five sides.

final Keyword In Java

- All methods and variables can be overridden by default in subclasses.
- Using the final keyword means that the value can't be modified in the future.
- Following are different contexts where final is used.



- **Final variables**
  - When a variable is declared with *final* keyword, its value can't be modified.
  - We must initialize a final variable, otherwise compiler will throw compile-time error.
  - Example,

    final int SIZE=10;
  - The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned.
  - Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

- **Final methods**

- When a method is declared with *final* keyword, it is called a final method.
- A final method cannot be overridden.
- Making a method final ensures that the functionality defined in this method will never be altered in any way.
- Constructors cannot be final.
- Example,

```
final void showData()
{
        ……….
}
```

- **Final class**

- When a class is declared with *final* keyword, it is called a final class.
- A final class cannot be extended(inherited).
  - One is definitely to prevent inheritance, as final classes cannot be extended.
  - For example, all Wrapper Classes like Integer, Float etc. are final classes.
- Example,

```
final class student
{
        …….
}
```

- **Example**

```
final class XYZ
{
        public void display()
        {
                System.out.println("This is a final method.");
        }
 }
class ABC extends XYZ // Compiler Error: cannot inherit from final XYZ
{
        void demo()
        {
                System.out.println("My Method");
        }
}
```

Interface

- Interface looks like a class but it is not a class.
- An interface can have methods and variables just like the class but the methods declared in interface are by default abstract.
- Also, the variables declared in an interface are public, static & final by default.
- To declare an interface, use the keyword interface.

The syntax is,

```
interface interface_name
{
 // declare constant variables
 // declare methods that abstract by default.
}
```

- Here interface is the keyword for declaring an interface.
- interface_name is the name of the interface.
- Variables are declared as follows,

static final data_type varname=value;

- Method declaration will contain only a list of methods without any body statements.
- Methods are declared as follows,

abstract return_type methnamae (parameter_list);

- **Example,**

```
interface item
{
  // public, static and final
        final static int NUM = 10;
  // public and abstract
         void display();
}
```

- Interfaces have the following properties –

- An interface is implicitly abstract. Do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Implementing interfaces

- Interfaces are used as super classes whose properties are inherited by classes.
- The format is,

```
class class_name implements interface_name
{
        Body of the class
}
```

  - Here the class class_name implements the interface interface_name.
  - The methods that implement an interface must be declared **public.**
  - **Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.**

Calculate Area of Rectangle using Interface- Example 1

```
interface area
{
                double pi = 3.14;
                double calc(double x,double y);
 }

class circle implements area
 {
 public double calc(double x,double y)
 {
                return(pi*x*y);
 }
 }
class  interface_test
   {
```

```
  public static void main(String arg[])
  {
  circle c = new circle();
   double a=c.calc(10.5,20.6);
    System.out.println("\nArea of Rectangle is : " + a);
   }
}
```

Example 2-

```
interface callback
{
        void back(int param);
}
class client implements callback
{
        public void back(int p)
        {
                System.out.println("back( ) method is called with "+p);
        }
}
class interface_test2
{
        public static void main(String arg[ ])       {
                callback obj =new client( );
                obj.back(42);
        }
}
```

Output

```
back( ) method is called with 42
```

Example 3

```
interface callback
{
```

```
        void back(int param);
}
class client implements callback
{
        public void back(int p)
        {
                System.out.println("back( ) method is called with "+p);
        }
}
class another implements callback
{
        public void back(int p)
        {
                System.out.println("back( ) method is called with "+ (p*p) );
        }
        void meth_another( )
        {
                 System.out.println("Only for class another");
        }
}

class interface_test2
{
        public static void main(String arg[ ])        {
        callback obj =new client( );
        obj.back(42);
        callback ob = new another( );
        ob.back(10);
        //ob.meth_another( );
        another ob1=new another();
        ob1. meth_another();
        }
}
```

Output

back( ) method is called with 42

back() method is called with 100

Only for class another

- In this program the variable obj is declared to be of the type interface "callback" type.
- Yet it is assigned an instance of class "client" type.
- Although obj can be used to access the back( ) method, it can't access any other members of client class.
- An interface reference variable only has the knowledge of members declared by its interface declaration.

Implementing multiple interfaces

- Unlike the singly inherited class hierarchy, we can include as many interfaces as we need in one class.
- That class will implement the combined behavior of all the included interfaces.
- To include multiple interfaces in a class, just separate their names with commas.
- The syntax is,

```
class class_name implements interface_name1, interface_name2,.....
{
Body of the class
 }
```

- **Example,**

```
interface item
{
        int code=1001;
        String name="Fridge";
}
interface item1
{
        void display( );
}
class Test implements item, item1
{

public void display()
```

```
                {
                System.out.println("Item code:"+code);
                System.out.println("Item Name:"+name);
                }
}

class interface_test3
{
            public static void main(String arg[ ])
            {
                Test obj =new Test( );
                obj.display();
            }
}
```

Output

```
Item code:1001
Item Name:Fridge
```

Extending Interfaces

```
interface inteface_name1
{
}
interface interface_name2 extends interface_name1
{
}
class class_name implements interface_name2
{
}
```

Packages

Packages are Java‘s way of grouping a variety of classes and/or interfaces together. The grouping is done according to functionality. Packages act as containers for classes It is a concept similar to class libraries in other languages Packages are organized in hierarchical manner.

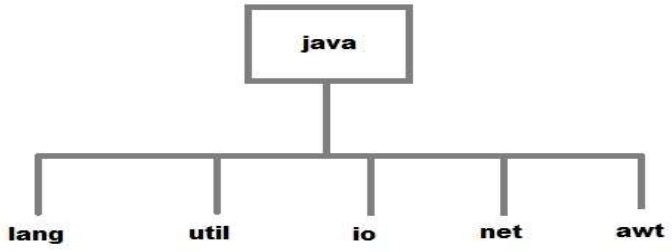
Packages provide a way to hide classes thus preventing other programs or packages from accessing classes that are meant for internal use only.

- In java packages can be classified into two.

  - **Built-in Packages**
  - **User Defined Packages**

Built-in Packages

These are packages which are already available in java language. These packages provide interfaces and classes, and then fields, constructors, and methods to be used in program. Built-in packages consist of a large number of classes which are a part of Java **API**. The following are the packages provided by Java



Some of the commonly used built-in packages are:

1) **java.lang:**

- lang stands for language.
- This package contains primary classes and interfaces essential for developing a basic java program.
- It consists of wrapper classes which are useful to convert primary data types into objects.
- There are classes like String, StringBuffer to handle strings. There is a Thread class to create individual processes.
- Runtime and system classes are also present in this package which contains methods to execute an application and find the total memory and free memory available in JVM.
- This is the only package that is automatically imported into every Java program.

2) **java.io:**

- Stands for input and output**.**

- A stream represents flow of data from one place to another place.
- Streams are useful to store data in the form of files and also to perform input-output related tasks

3) **java.util:**

- Contains utility classes which implement data structures like Linked List, Dictionary and support for Date / Time operations.

4) **java.applet:**

- Contains classes for creating Applets.

5) **java.awt:**

- Stands for abstract window toolkit
- Contain classes for implementing the components for graphical user interfaces such as windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields.

6) **java.net:**

- Contain classes for supporting networking operations.

Import Java Package

☐ To import java package into a class, we need to use **import** keyword which is used to access package and its classes into the java program.

☐ **Import a Class**

- To import classes from a package use the keyword import.
- The syntax is,

```
import package_name . classname;
```

- For exampel,

```
import java.util.Date;
import java.lang.Math;
```

☐ **Import all classes of the package**

- We can import all the classes inside a package.
- The syntax is,

```
import packagename . *;
```

For example,

```
import java.io.*;
```

Example program

```
import java.lang.Math;
 // This program can now use all things defined inside
 // the class java.lang.Math without
class MyProgram
{
        public static void main(String[] args)
        {
         double a;
         a = Math.sqrt(2.0);
         System.out.println(a);
         }
}
```

- **Other methods inside the Math class**

  pow( )
  max( )
  sin( )
  cos( )

User Defined Packages

Just like built-in packages users of java can create their own packages. They are called user defined packages. The user can create their own package using the keyword **package**. This is the general form of the package statement:

package pkg1;

- Here **pkg1** is the name of the package.
- This must be the first statement in a java source file.
- For example

```
package firstPackage;
public class FirstClass
{
        ……
        ……..
```

```
}
```

- Here package name is firstPackage.
- The class FirstClass is now considered a part of this package.
- This would be saved as a file called **FirstClass.java**
- Any classes declared within that file will belong to the specified package.
- The package statement defines a name space in which classes are stored.
- The visibility mode or access specifier of a class declared inside the class must be public.
- The file FirstClass.java should be saved in the directory named firstPackage, which is the user defined package.
- On compiling java will create a class file and store it in the same directory.
- Remember that the class file must be located in a directory that has the same name as package and this directory should be a sub directory of the directory where classes that will import the packages are located.
- More than one classes can include the same package statement and each class is saved as a separate java file.

Example program for package

```
package p1;
public class MyCalculator
{
        public int Add(int x,int y)
        {
                return x+y;
        }
        public int Subtract(int x,int y)
        {
                return x-y;
        }
        public int Product(int x,int y)
        {
```

```
            return x*y;
        }
}
```

- **Steps to execute the above package example**
  - Create a subfolder named **p1 inside our directory where the main source files are stored.**
  - **Save the above program as MyCalculator.java** inside **p1** folder, where MyCalulator is the public class of package p1.
  - Compile the java file Mycalculator.java. This will create the .class file in the directory p1.

Example for the source file which is import the package p1

```
import p1.MyCalculator;
class TestPackage
        {
        public static void main(String[] args)     {
                MyCalculator M = new MyCalculator();
                int s=M.Add(45,10);
                int d=M.Subtract(45,10);
                int p=M.Product(45,10);
                System.out.print("\n\n\tThe Sum is : " +s);
                System.out.print("\n\n\tThe Subtract is :" +d);
                System.out.print("\n\n\tThe Product is :" +d);
        }
}
```

Steps to execute the above java source program

- Save this program as TestPackage.java inside the main directory.
- Compile and Run **TestPackage.java**

- **The output is**

  The Sum is : 55

  The Subtract is : 35

  The Product is : 450

- During compilation of **TestPackage.java the compiler checks for file** MyCalulator**.class in p1 directory for information it needs, but it does not include the code from** MyCalulator**.class in the file TestPackage.java**

Package with Multiple Public Classes

We can add multiple classes inside one package. If we want to create multiple public classes under one package, we have to store them in **separate source files** and attach the **package** statement as the first statement in those source files.

Example program for multiple classes inside one package

```
package mypack;
public class Box
 {
        int length,breadth,height;
        public Box(int l,int b,int h)
         {
         length=l;
         breadth=b;
         height=h;
        }
         public int vol()
         {
        return length*breadth*height;
          }
}
```

- **Save this program as Box.java under the subdirectory mypack**

- **Compile it and it will create Box.class inside mypack folder**

```
package mypack;
public class circle
  {
          int r;
         public circle(int rr)
          {
                r=rr;
         }
         public double area()
          {
                return (3.14*r*r);
          }
}
```

- **Save this program as circle.java under the subdirectory mypack**
- **Compile it and it will crate circle.class inside mypack directory.**

```
import mypack.*;
class MultiPackage
         {
         public static void main(String[] args)    {
                Box ob1 = new Box(10,20,30);
                circle ob2=new circle(12);
                int v=ob1.vol( );
                double a=ob2.area( );
                System.out.print("\n\n\tThe Volume is : " +v);
                System.out.print("\n\n\tThe area is :" +a);
          }
 }
```

- ☐ Save this program as MultiPackage.java inside the main directory.
- ☐ Compile and Run MultiPackage**.java**
- ☐ **The output is**

The volume is: 600

The area is: 452.16

Exception Handling

- There are basically 3 types of errors in a java program:
  - § Compile-time errors
  - § Run-time errors
  - § Logical errors.

Compile-time errors

- ❑ compile time errors are syntactical errors found in the code, due to which a program fails to compile.
- ❑ For example, forgetting a semicolon at the end of a java program, or writing a statement without proper syntax will result in compilation-time error.

Run-time errors

- Run time errors represent inefficiency of the computer system to execute a particular statement, means computer system cannot process.
- For example, division by zero error occur at run-time.

```
class Rerror
{
        public static void main(String args[ ])    {
            int a=10,b=0;
            System.out.println("a/b: "+(a/b));
        }
}
```

Logical Errors

- Logical errors depict flaws in the logic of the program and they are identified on observing the output .
- Logical errors are not detected either by the java compiler or JVM.

Exception

- An Exception in java signals an error or an unusual event that happens during the execution of a program.
- One important point to note is that all errors in your program are not signaled by exceptions.
- An exception arises in a code sequence at run time.
- In other words, an exception is a run-time error.
- The ability of a program to intercept run-time errors, take corrective measures and continue execution is referred to as exception handling.
- Exception handling separates the code that deals with errors from the code that is executed when program is normally executing.
- It also provides a way of enforcing a response to particular errors.
- An exception in java is an object that is created when an abnormal situation arises in our program.
- This exception object has fields that store information about the nature of the problem.
- When the error situation is encountered we will say an exception is thrown.
- That is the object identifying the exceptional circumstances is tossed as an argument to the program code that has been written specifically to deal with that kind of problem.
- The code receiving the exception object as a parameter is said to catch it.

Uncaught Exceptions

- Consider the following program,

```
class UncaughtExe
 {
                public static void main(String args[ ])
```

```
        {
            int i=10;
            int j=10;
            int d=i/(i-j);
            int s=i+j;
            System.out.println("Division =" +d);
            System.out.println("Sum=" +s);
        }
}
```

- Here when the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.
- This causes the execution of the class UncaughtExe to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
- In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- At the Compilation time the above program will not cause any errors.
- But On running the program the Output obtained will be:

  Exception in thread "main" java.lang. ArithmeticException: / by zero at UncaughtExe.main(UncaughtExe.java:9)

- Here type of the exception thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened.

  Exception in thread "main" java.lang.ArithmeticException: / by zero

  - this is the string describing the exception

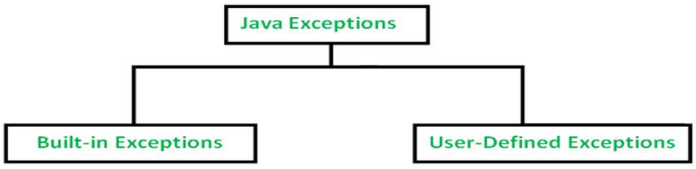
- The stack trace will always show the sequence of method invocations that led up to the error.
- The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

at UncaughtExe.main(UncaughtExe.java:9)

- this is the StackTrace

Types of Exception in Java

- Two types of exceptions in java:
  - Built-in Exceptions
  - User-Defined Exceptions
- Java defines several types of exceptions that relate to its various class libraries. These are Built-in Exceptions
- Java also allows users to define their own exceptions. These are known as User-defined Exceptions



Built-in Exceptions

- Built-in exceptions are the exceptions which are available in Java libraries.
- These exceptions are suitable to explain certain error situations.
- Below is the list of important built-in exceptions in Java.

1. ArithmeticException

It is thrown when an exceptional condition has occurred in an arithmetic operation.

2. ArrayIndexOutOfBoundsException

It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

3. ClassNotFoundException

This Exception is raised when we try to access a class whose definition is not found

4. FileNotFoundException

This Exception is raised when a file is not accessible or does not open.

5. IOException

It is thrown when an input-output operation failed or interrupted

7. NoSuchFieldException

It is thrown when a class does not contain the field (or variable) specified

8. NoSuchMethodException

It is thrown when accessing a method which is not found.

9. NumberFormatException

This exception is raised when a method could not convert a string into a numeric format.

User-Defined Exceptions

- Sometimes, the built-in exceptions in Java are not able to describe a certain situation.
- In such cases, user can also create exceptions which are called 'user-defined Exceptions'.

Exception Handling in Java

- Java exception handling is managed via five keywords:
  - try, catch, throw, throws, and finally

1. try-catch block

- Syntax of Java try-catch

```
try
{
        //code that may throw an exception
 }
   catch(Exception_class_Name obj)
   {
```

```
        // Block of code to handle exception
    }
```

- Following figure shows the working of try-catch block

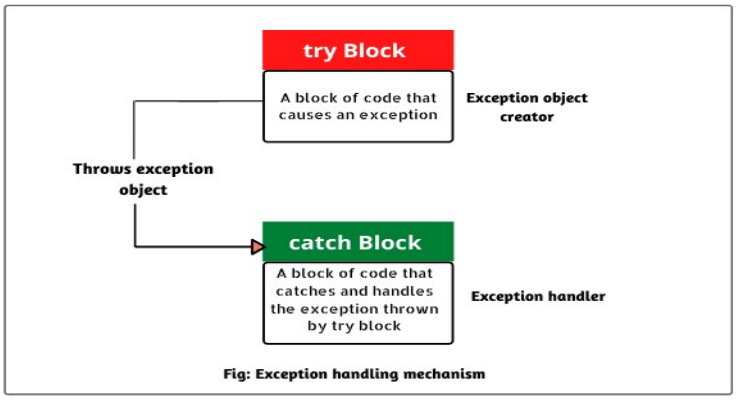


Fig: Exception handling mechanism

try block

- The "try" keyword is used to specify a block where we should place exception code.
- Java **try** block is used to enclose the code that might throw an exception.
- It must be used within the method.
- If an exception occurs at the particular statement of try block, the rest of the block code will not execute.
- So, it is recommended not to keeping the code in try block that will not throw an exception.
- The try block must be followed by either catch or finally.
- It means, we can't use try block alone.

catch block

- A catch block is where we handle the exceptions, this block must follow the try block.
- A catch block must specify what type of exception it will catch.
- A catch clause that specifies the exception type that we wish to catch.
- Java catch block is used to handle the Exception by declaring the type of exception within the parameter

- The declared exception must be the parent class exception (i.e., Exception) or the generated exception type.
- However, the good approach is to declare the generated type of exception.
- The catch block must be used after the try block only.

- **Example 1**

```
class Exe_sample
{
public static void main(String args[ ])
{
int i=10;
int j=10;
                try
                {
                        int d=i/(i-j);
                        System.out.println("Division =" +d);

                }
                catch(ArithmeticException eobj)
                {
                        System.out.println("Exception type: "+eobj);
        }
System.out.println("After try-catch");
}
}
```

Output

```
Exception type: java.lang.ArithmeticException: / by zero
After try-catch
```

Example 2

```
class Exe_sample1
{
public static void main(String args[ ]){
```

```
int i=10;
int j=10;
                try
                {
                        int s=i+j;
                        System.out.println("Sum=" +s);
                        int d=i/(i-j);
                        System.out.println("Division =" +d);

                }
                catch(ArithmeticException eobj)
                {
                        System.out.println("Exception type: "+eobj);
                }
System.out.println("After try-catch");
}
}
```

Output

```
Sum=20
Exception type: java.lang.ArithmeticException: / by zero
After try-catch
```

Using Multiple catch Blocks

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, we can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

Example program

```
class MultiCatch
{
   public static void main(String args[])     {
```

```
    try      {
                 int a = args.length;
                System.out.println("a = " + a);
                int b = 42 / a;
                int c[ ] = { 11,22};
                c[42] = 99;
       }
       catch(ArithmeticException e)
       {
                System.out.println("Divide by 0: " + e);
        }
     catch(ArrayIndexOutOfBoundsException e)
    {
                System.out.println("Array index: " + e);
    }
  System.out.println("After try/catch blocks.");
 }
}
```

- ✓ This program will cause a division-by-zero exception if it is started with no command-line parameters, since 'a' will equal zero.
- ✓ It will survive the division if you provide a command-line argument, setting 'a' to something larger than zero.
- ✓ But it will cause ArrayIndexOutOfBoundsException, since the int array c has a length of 2, yet the program attempts to assign the value 99 to the index 42.

Output

First run

```
C:\Myfolder>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

Second run

```
C:\Myfolder>java MultiCatch Saritha
a = 1
Array index: java.lang.ArrayIndexOutOfBoundsException
```

After try/catch blocks.

Nested Try statements

- The try statement can be nested. That is, a try statement can be inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack.
- If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement‘s catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception.

Example,

```
classNestTry
 {
    public static void main(String args[ ])     {
    try     {
                int a = args.length;
                /* If no command-line args are present, a divide-by-zero exception is generated.
                */
                int b = 42 / a;
                System.out.println("a = " + a);
    try      {
                 // nested try block
                /* If one command-line arg is used then a divide-by-zero exception will be
                generated by the following code. */
                 if(a==1)
                    a = a/(a-a); // division by zero

                 if(a==2)

                {
                   int c[ ] = { 11,22 };
                   c[42] = 99;
                // generate an out-of-bounds exception
```

```
                }
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println("Array index out-of-bounds: " + e);
    }
 }
catch(ArithmeticException e)
 {
   System.out.println("Divide by 0: " + e);
  }
}
}
```

- This program nests one try block within another.
- The program works as follows.
  - When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block.
  - Execution of the program by one command-line argument generates a divide-by-zero exception from within the nested try block.
  - Since the inner block does not catch this exception, it is passed onto the outer try block, where it is handled.
  - If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.
  - Here are sample runs that illustrate each case:

First run

```
C:\Myfolder>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
```

Second run

```
C:\Myfolder>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
```

Third run

```
 C:\ Myfolder >java NestTry One Two
 a = 2
Array index out-of-bounds:  java.lang.ArrayIndexOutOfBoundsException
```

2. throw

- In java we can create our own exception class and throw that exception using throw keyword.
- These exceptions are known as **user-defined** or **custom** exceptions. The Java throw keyword is used to explicitly throw an exception.
- The throw keyword is mainly used to throw custom exception.
- The throw keyword can also be used to throw built-in exceptions
- **Syntax of throw keyword:**

```
throw new exception_class("error message");
```

Example pgm 1.

- In this example, we have created the method validate( ), that takes integer value as a parameter.
- If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote

```
public class TestThrow1
{
  static void validate(int age)
   {
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
   }
  public static void main(String args[])   {
      validate(13);
     System.out.println("rest of the code...");
  }
}
```

Output

Exception in thread main java.lang.ArithmeticException:not valid

3. throws

- If a method is capable of causing an exception that it does not handle.
- The general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
 {
        body of method
}
```

- Here, exception-list is a comma-separated list of the exceptions that a method can throw.

4. finally

- A finally block must be associated with a try block, we cannot use finally without a try block.
- finally statement can be used to handle an exception that is not caught by any of the previous catch block.
- It may be added immediately after the try block or after the last catch block.

- The general forms are:

1)
```
try
{
  .......
}
finally
{
.........
}
```

2)
```
try
{
  .......
}
catch(Exception obj)
{
   ...........
}
```

```
catch(Exception obj)
{
 .....
}
finally
{
.....
}
```

User Defined Exceptions

- In the standard package java.lang, java defines several exception classes.
- The base class for all built-in exceptions are Exception.
- Sometimes the built-in exceptions are not able to describe a certain situation.
- In such cases it is possible for the user to create his own exceptions which are called User-Defined Exceptions.
- The following steps are followed in the creation of User-Defined Exceptions.
- Since all Exceptions are subclasses of Exception class, So user should create a sub class of exception.

```
class MyException extends Exception
```

  - User can create an empty constructorin his own exception class. he can use it ,in case he doesn‘t want to store any exception details. If the user doesn‘t want to create an empty object to this exception class ,we can eliminate writing the default constructor.

```
MyException()
{ }
```

  - The user can create a parameterized constructor with a string as a parameter. We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

```
MyException(String str)
{ super(str); }
```

  - When the user want to raise his own exception, he should create an object to his exception class and throw it using throw clause as:

```
MyException me=new MyException(“Exception Details”);
 throw me;
```

- **Example pgm**

```
// This program creates a custom exception type.
class MyException extends Exception
```

```
 {
                private int detail;
                MyException(int a)
                { detail = a; }
                public String toString()
                {
                        return "MyException[" + detail + "]";
                }
}

class ExceptionDemo
{
                static void compute(int a) throws MyException
                 {
                System.out.println("Called compute(" + a + ")");
                if(a > 10)
                        throw new MyException(a); System.out.println("Normal exit");
}
public static void main(String args[ ])       {
                  try   {
                                compute(1);
                                compute(20);
                        }
                        catch (MyException e)
                        {
                                System.out.println("Caught " + e);
                        }
                  }
}
```

- This example defines a subclass of Exception called MyException.
- This subclass has only a constructor and an overridden toString( ) method that displays the value of the exception.
- The ExceptionDemo class defines a method named compute( ) that throws a MyException object.
- The exception is thrown when compute( )'s integer parameter is greater than 10.

- The main( ) method sets up an exception handler for MyException, then calls compute( ) with a legal value (less than 10) and an illegal one to show both paths through the code.
- Here is the Output is:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Example pgm 2

```
class MyException extends Exception
{
                int ex;
               MyException(int a)
                {   ex=a;  }
              public String toString()
               {   return "MyException[" + ex +"] is less than zero";
               }
}
class Test
{
              static void sum(int a,int b) throws MyException
               {
                if(a<0)
                {
                  throw new MyException(a); //calling constructor of user-defined exception
                }
                else
                {
                        System.out.println(a+b);
                 }
              }

public static void main(String args[ ])
 {
               try
               {
```

```
                sum(-10, 10);
            }
            catch(MyException me)
            {
                System.out.println(me); //it calls the toString() method of user-defined
            Exception
            }
  }
}
```

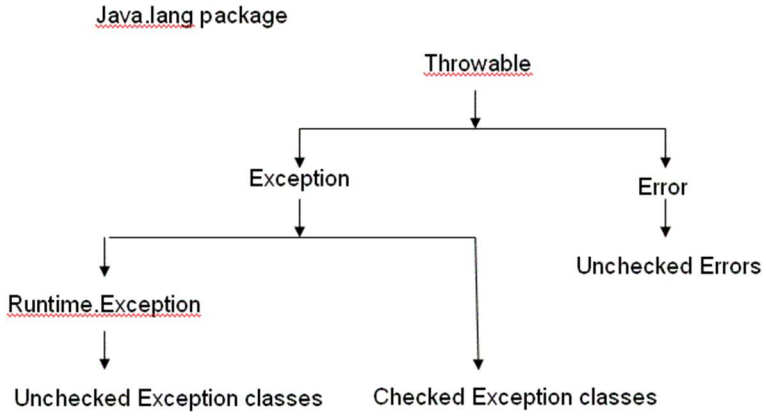
Output

MyException[-10] is less than zero

- Here we override the toString() function, to display customized message.
- The keyword **"throw"** is used to create a new Exception and throw it to the catch block.

Types of exceptions

- There are mainly two types of exceptions:
  - Checked
  - Unchecked.
- Unchecked exceptions come in two types:
  - Errors
  - Runtime exceptions
- The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses:
  - Exception and
  - Error.
- A hierarchy of Java Exception classes are given below:



Checked Exceptions:

- These are the exceptions that are checked at compile time.
- If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.
- Checked exceptions are the type that programmers should anticipate and from which programs should be able to recover.
- All Java exceptions are checked exceptions except those of the Error and RuntimeException classes and their subclasses.

- A checked exception is an exception which the Java source code must deal with, either by catching it or declaring it to be thrown.
- Checked exceptions are generally caused by faults outside of the code itself - missing resources, networking errors, and problems with threads come to mind.
- These could include subclasses of

  FileNotFoundException, UnknownHostException, etc

Unchecked Exceptions:

- **Unchecked** are the exceptions that are not checked at compiled time.
- The classes which inherit RuntimeException are known as unchecked exceptions e.g.

  ArithmeticException,

  NullPointerException,

ArrayIndexOutOfBoundsException etc.

- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

- All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class. Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment.
- Example: JVM is out of memory.
- Normally, programs cannot recover from errors. **Error** are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error. The Exception class has two main subclasses: IOException class and RuntimeException Class.